Randomization in Algorithms & Data Structures
Part 1

Ibrahim Albluwi

Las Vegas Algorithms.

Randomized algorithms that always give the correct results, but whose running time can vary (because of the use of randomness). Useful Las Vegas algorithms have good *expected* running times.

The most obvious example is *quicksort*, whose running time falls between $\Omega(n \log n)$ and $O(n^2)$. We can randomize the algorithm by shuffling the array before running quicksort. This does not affect the correctness of the algorithm but makes the *expected* running time $\Theta(n \log n)$.

Another example is *quickselect*, whose running time is bounded by $\Omega(n)$ and $O(n^2)$. Randomly shuffling the array makes the *expected* running time $\Theta(n)$.

Monte Carlo Algorithms.

Randomized algorithms that have a deterministic running time, but are not guaranteed to produce the correct solution (because of the use of randomness). Useful Monte Carlo algorithms are efficient and provide a correct solution with high probability or an approximate solution that is very close to the correct solution.

Example. Given n points in the 2D plane, check if it is possible to draw a line that passes through at least $n/8$ of them.

Inefficient Solution. For each pair of points, create a line passing through the pair and count how many other points pass through this line. Report true if any of the lines passes through $n/8$ of the points. There are $\Theta(n^2)$ pairs of points and checking a line requires $\Theta(n)$ time, so the total is $\Theta(n^3)$.

Monte Carlo Algorithm. Pick a random pair of points. Report *true* if the line passing through them also passes through $n/8$ other points. Report *false* otherwise.

Analysis. If there are $n/8$ collinear points, there is at least a $\frac{1}{8} \times \frac{1}{8} = \frac{1}{64}$ chance that the first randomly picked point *and* the second randomly picked point both are on the line passing through these collinear points. I.e. this algorithm has a $1 - \frac{1}{64} = \frac{63}{64} = 98.4375\%$ chance of failing.

We will modify the algorithm such that it repeats the above 400 times, and reports failure if all of the 400 iterations did not find a line passing through $n/8$ points. If there are $n/8$ collinear points, then the modified algorithm has a chance of failure of $(0.984375)^{400} \approx 0.184\,\%$, which is close to zero!
If $n = 100{,}000$ then testing only 400 pairs of points, instead of $\frac{1}{2}(100000 \times 99999) = 4999950000$ pairs is a big win, especially since the result is expected to be correct around 99.8% of the time. Note that We can reduce the probability of failure even further by increasing the number of iterations.

A Useful Trick.
If the probability that A is incorrect = 99% then repeating A 450 times makes the probability that all the runs are incorrect $= 0.99^{450} \approx 1\%$

We will discuss in this lesson *data structures* that are *expected* to provide a correct answer or an approximation (as in Monte Carlo algorithms) or that are *expected* to answer queries efficiently (as in Las Vegas algorithms).

Bloom Filters.

> **Problem**. Given n elements (where n is large), answer queries of the form: Was element k seen before?
> **Requirement.** Be as memory efficient as possible.
> **Assumption.** A small possibility of incorrect `YES` answers is acceptable.

Assume that you have a server that receives a large number of requests. You would like to quickly identify requests from returning customers (to give them special treatment). However, accessing the database is costly, so you want to check the database only if there is a high chance that you have seen the customer before. Hence, you want a memory-efficient way that can act as a filter against unnecessary database accesses.

This is a recurring problem in many domains and can be reformulated as the problem of providing a memory-efficient implementation for an abstract data type with the following operations:

`INSERT(x):`	Adds x to the set.
`CONTAINS(x):`	Returns `FALSE` only if x is not in the set. Returns `TRUE` if x is very likely to be in the set.

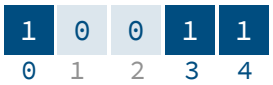
Using a hash table guarantees that **`CONTAINS`**`(x)` returns `FALSE` if and only if x is not in the set and `TRUE` if and only if it is in the set. This is consistent with the ADT requirements (very likely ≡ 100%). However, this is not memory efficient and is more than what the ADT requires (we are fine with a probability of say 99%).

Solution # 1. Use a hash table, but don't bother to resolve collisions!

Consider a hash function $h(x)$ that produces values in the range $[0,\ m)$. Assume also that `TABLE` is a bit vector of size m, where m is a very small number and `TABLE` is initialized to zeroes. The implementation of the ADT is as follows:

`INSERT(x):`	Set `TABLE[`$h(x)$`]` to **`1`**.
`CONTAINS(x):`	Return `TABLE[`$h(x)$`] == 1`

Example. Assume $h(x) = (3x + 1)\ \%\ m$, where $m = 5$, and we insert the elements $\{1,\ 3,\ 4\}$.
This gives the following table:



Explanation:

$h(1)$	$=$	$(3\times1+1)\,\%\,5$	$=$	$4\,\%\,5$	$=$	4
$h(3)$	$=$	$(3\times3+1)\,\%\,5$	$=$	$10\,\%\,5$	$=$	0
$h(4)$	$=$	$(3\times4+1)\,\%\,5$	$=$	$13\,\%\,5$	$=$	3

`INSERT(1)` sets index `4` to `1`.
`INSERT(3)` sets index `0` to `1`.
`INSERT(4)` sets index `3` to `1`.

Using the `TABLE`, we can answer membership queries as follows:

`CONTAINS(3)` returns `TRUE`	because	`TABLE[`$h(3) = 0$`]`	is set to `1`.	*(correct answer)*
`CONTAINS(5)` returns `FALSE`	because	`TABLE[`$h(5) = 16\ \%\ 5 = 1$`]`	is set to `0`.	*(correct answer)*
`CONTAINS(6)` returns `TRUE`	because	`TABLE[`$h(6)\,\%\,5 = 19\,\%\,5 = 4$`]`	is set to `1`.	*(wrong answer)*

Hence:
- If `TABLE[`$h(x)$`]` is set to `0`: x definitely was not seen before (otherwise, it would have been set to 1).
- If `TABLE[`$h(x)$`]` is set to `1`: x was seen before or another element y was seen before, where $h(x) = h(y)$.

Analysis of Solution # 1.

Assuming that $h(x) = i$, we are interested in knowing the probability that `TABLE[i]=1` after n insertions into the set. This probability represents how likely our solution will return `TRUE` regardless of whether the element was seen before or not.

Assuming that $h(x)$ produces values that are uniformly distributed in the range $[0, m)$, then after 1 insertion:

- The probability that `TABLE[i]=1` is $\frac{1}{m}$.
- The probability that `TABLE[i]=0` is $1-\frac{1}{m}$.

After n insertions, the probability that `TABLE[i]` is still set to `0` is $(1-\frac{1}{m})^n$. This can be simplified as follows:

$$(1-\tfrac{1}{m})^n = ((1-\tfrac{1}{m})^m)^{\frac{n}{m}}$$

$$= (e^{-1})^{\frac{n}{m}} = e^{-\frac{n}{m}} \qquad \text{knowing that } \lim_{x\to\infty}(1-\tfrac{1}{x})^x = e^{-1}$$

Hence, the probability that `TABLE[i]=1` after n insertions is $1 - e^{-\frac{n}{m}}$.

This probability is high regardless of the value of m. To make this more concrete, the following are some examples of error probabilities assuming $n = 10000$:

m	probability of error
100	≈ 1
1000	≈ 1
2000	≈ 0.993
5000	≈ 0.865
10,000	≈ 0.632
50,000	≈ 0.181
100,000	≈ 0.095
500,000	≈ 0.02
1000,000	≈ 0.01

In other words, this solution is highly likely to answer `TRUE` incorrectly, unless we use around $100n$ bits. Is this memory efficient?

To understand the memory efficiency of this solution, let's analyze the memory requirements of a hash table that uses separate chaining to store n integer keys. Assuming that the table has m linked lists and $m = n/4$, a C++ implementation of this table requires at least the following number of bits:

- 32 bits (integer key) + 64 bits (next pointer) for each key in the table.
- 64 bits for each linked list object.

The total is $(32 + 64)n + \frac{64}{4}n = 112n$ bits, which is very close to what our solution provided. I.e. our solution does not provide memory gains unless the stored keys are not integers and require much more than 32 bits.

Can we use the useful (repetition) trick we learned before to reduce the probability of error for a certain value of m that has a high probability of error?

Solution # 2. Use $k > 1$ hash functions instead of one (A.K.A. Bloom Filter)

Consider the (independent) hash functions $h_1(x)$, $h_2(x)$, …, $h_k(x)$ each producing values in the range $[0, m)$. Consider also the bit vector `T` of size m. We will modify the implementation of the ADT as follows:

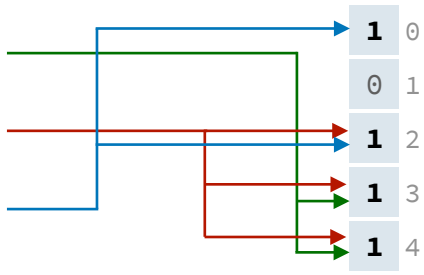
`INSERT(x):` Set `T[`$h_1(x)$`] = 1`, `T[`$h_2(x)$`] = 1`, … `T[`$h_k(x)$`] = 1`

`CONTAINS(x):` Return `T[`$h_1(x)$`] = 1` **`AND`** `T[`$h_2(x)$`] = 1` … **`AND`** `T[`$h_k(x)$`] = 1`

Example. $k = 3$, $h_1(x) = 3x \,\%\, m$, $h_2(x) = 4x \,\%\, m$, $h_3(x) = (2x + 2) \,\%\, m$.

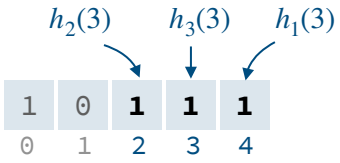
Assume that $m = 5$, and we insert the elements $\{1, 3, 5\}$. This gives the following table:

$h_1(1) = 3$	$h_2(1) = 4$	$h_3(1) = 4$	`INSERT(1)`
$h_1(3) = 4$	$h_2(3) = 2$	$h_3(3) = 3$	`INSERT(3)`
$h_1(5) = 0$	$h_2(5) = 0$	$h_3(5) = 2$	`INSERT(5)`

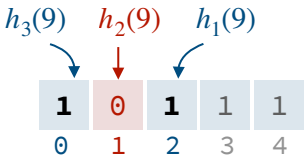


Using the bit vector, we can answer membership queries as follows:

`CONTAINS``(3)` correctly returns `TRUE` because all the bits are set to 1



`CONTAINS``(9)` correctly returns `FALSE` because **not** all the bits are set to `1`



If one of the hash functions does not lead to a 1-bit, then the element definitely was not seen before. However, even if all the hash functions lead to a 1-bit, it is only likely that the element was seen before (the bits could have been set by other elements). Try with `contains(6)`.

Analysis of Solution # 2.

Using the same logic used in the previous analysis, and assuming that the hash functions are independent, we have the following probabilities:

- The probability that a certain bit is still `0` after n insertions using k hash functions = $(e^{-\frac{n}{m}})^k = e^{-\frac{kn}{m}}$
- The probability that a certain bit is set to `1` after n insertions using k hash functions = $1 - e^{-\frac{kn}{m}}$

Thus, the probability that all k bits are set to 1 after n insertions is:

$$\epsilon = (1 - e^{-\frac{kn}{m}})^k$$

We can see that increasing the number of hash functions k can:

- *Decrease* the probability of error (intuition: you need to satisfy more hash functions before answering `YES`)
- *Increase* the probability of error (intuition: there is a higher probability that a bit will be set to `1`).

Fixing n and m, we find (using differentiation) that the optimal value of k that reduces the probability of error:

$$k = \frac{m}{n} \ln 2$$

Substituting the optimal k in ϵ we get:

$$(1 - e^{-\frac{\frac{m}{n} \ln 2 \times n}{m}})^{\frac{m}{n} \ln 2}$$

$$= (1 - e^{-\ln 2})^{\frac{m}{n} \ln 2}$$

$$= (\frac{1}{2})^{\ln 2 \frac{m}{n}} \approx 0.69314^{\frac{m}{n}}$$

Is this probability of error good? Let's look at concrete values assuming $n = 10{,}000$ and the optimal value of k is used:

m	optimal k	probability of error
10,000	< 1	≈ 0.67
20,000	≈ 1.386	≈ 0.48
30,000	≈ 2.079	≈ 0.33
40,000	≈ 2.773	≈ 0.23
50,000	≈ 3.466	≈ 0.16
100,000	≈ 6.931	≈ 0.026
125,000	≈ 8.664	≈ 0.01

This means that if we have 10,000 elements, we can get an error probability of $\sim 1\%$ using $8 - 9$ hash functions and 125,000 bits (which is only ~ 0.11 of the required space for a hash table of integers as calculated before). Memory savings can be even greater if the keys are objects that require more than 32 bits.

The above probability of error calculations are crude (although useful) estimates. They assume that the hash functions are independent, which is usually not true in reality. They also assume that the optimal k value can always be used, which is not true, because k has to be an integer but $\frac{m}{n} \ln 2$ is not always an integer. There are better error estimates that you can find, but they are outside the scope of our discussion.

> **Useful Tools**. There are simple online calculators for calculating the optimal values for k and m given a certain number of elements n and a desired probability of error. The following are examples:
> - https://hur.st/bloomfilter/
> - https://krisives.github.io/bloom-calculator/

This data structure is called a **Bloom Filter** and is widely used. For example, Google Chrome stores a small bloom filter on your machine that can quickly answer queries on whether a URL is safe or probably malicious. This is faster than making a request and is more memory efficient than storing the whole database of unsafe URLs on your machine.

Cardinality Estimation.

> **Problem**. Count the number of *unique* elements in a data stream.
> **Requirement.** Use $O(\log n)$ space, where n is the number of elements in the stream seen so far.
> **Assumption.** An approximate count is acceptable.

A straightforward solution is to insert every new element into a set (implemented as a hash table or a balanced binary search tree), which makes the size of the set (at any given point in time) the exact count of the number of unique elements seen so far. However, this solution requires space proportional to the number of unique elements in the data stream, which violates the memory requirements of the problem.

This is a common problem, where we cannot afford to store all of the data, but need to answer queries that seem to require storing all of the data. One example is counting the number of unique visitors to a website. Storing an identifier for every single visitor can be prohibitively impractical. However, without storing an identifier for each visitor, how can we know if the new visitor is not a returning visitor?

Solution # 1. Probabilistic Counting

Assume that we have a box containing 1110 balls with the following colors:

	Probability of being drawn randomly
1000 blue balls	$1000 \,/\, (1000 + 100 + 10) \approx 90\,\%$
100 green balls	$100 \,/\, (1000 + 100 + 10) \approx 9\,\%$
10 red balls	$1 \,/\, (1000 + 100 + 10) \approx 1\,\%$

The probabilities tell us that we might see a blue ball after drawing randomly from the box one or two times only (because $\frac{1}{0.9} = 1.11$) and that we might see a green ball after drawing randomly from the box around 11 times ($\frac{1}{0.09} = 11.11$). Similarly, we might need to draw randomly around 100 times before we see a red ball ($\frac{1}{0.01} = 100$). Generally speaking, if event A occurs with probability p, then we should expect to encounter A after $1/p$ trials. The key insight here is that rare events need a lot of time before they are observed.

Assume you would like to estimate how many times someone drew randomly from the box. Knowing that the person drew a red ball gives you a hint that the number of trials is probably 100 or more. This rough way of estimation can be used as a basis for an algorithm to count the unique elements in a data stream as follows:

```
m = 0                                   # m is short for 'maximum'
FOREACH x in the stream:
      h = HASH(x)                       # 32-bit integer hash value of x
      c = COUNT-TRAILING-ONES(h)        # e.g. 1010111 has 3 trailing 1s
      m = MAX(m, c)                     # update m if c > m

RETURN 2^m
```

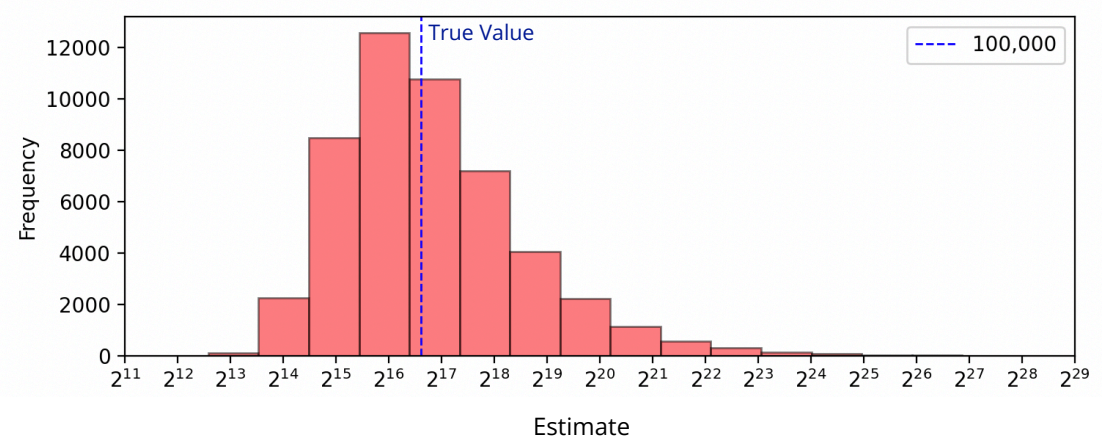
How frequent are integers whose binary representation have many trailing ones? Looking at the table below, we see that ½ of the integers end with 1, ¼ end with two 1s, ⅛ end with three 1s, etc. I.e. integers with many trailing ones are rare and encountering one of them can be a hint that we have seen many random integers.

All possible hashes assuming 4-bit integer hash values
0000
000**1**
0010
00**11**
0100
010**1**
0110
0**111**
1000
100**1**
1010
10**11**
1100
110**1**
1110
1111

Number of trailing 1's	Probability	Expected trials before the first encounter
1	$\frac{1}{2}$	$2^1 = 2$ (or 1/0.5)
2	$\frac{1}{4}$	$2^2 = 4$ (or 1/0.25)
3	$\frac{1}{8}$	$2^3 = 8$
4	$\frac{1}{16}$	$2^4 = 16$

This observation on the frequency of trailing ones is used by the above algorithm to estimate the number of unique elements in the stream. The algorithm counts the number of trailing ones in the hash values of the elements instead of the elements themselves. This is to get rid of any patterns in the stream (e.g. if the stream is made of even numbers only, all the values will have zero trailing 1s !). The maximum encountered number of trailing 1s (called m in the algorithm) is used to estimate the number of unique elements encountered. This algorithm was proposed by Flajolet and Martin in 1983.

How good is this estimate? To answer this question, we performed 50,000 simulations. In each simulation, we generated 100,000 random integers (representing the data stream). We used the above algorithm to estimate the number of unique integers in the stream and plotted the following histogram. The x-axis is the estimate made by the algorithm, and the y-axis is how many times the estimate was made.



As is clear from the plot, the estimates vary wildly around the true value ($100{,}000 \approx 2^{16.61}$), The range of the made estimates goes from 2^{13} ($\approx 8{,}000$) to around 2^{27} ($\approx 67{,}000{,}000$). Nevertheless, the estimates seem to fluctuate around the true value, where the peak of the histogram leans slightly below the true value. This is why Flajolet and Martin added a correction factor to the algorithm, making the estimate $\frac{2^m}{0.77351}$ instead of 2^m. This correction factor was analyzed theoretically and validated experimentally.

The above simulation shows that the estimate is excellent on average. However, this is not very useful in practice. Think of a calculator that sometimes estimates 5×5 to be 20 and sometimes estimates it to be 30. The average estimation of the calculator is excellent ($(20 + 30)/2 = 25$), but the calculator cannot be reliably used, because it always gives estimates that are 20 % off. I.e., our algorithm has excellent estimation on average but deviates frequently (and by a large margin) from the true value, making it an unreliable estimator.

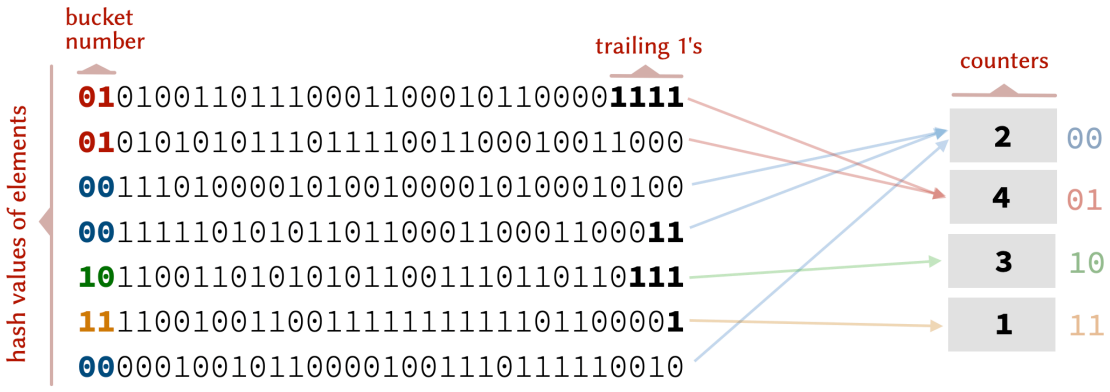
How can we reduce the variance of our estimates? Can we use the useful (repetition) trick we learned before?

Solution # 2. Probabilistic Counting with Stochastic Averaging (PCSA), LogLog, and Hyper-LogLog

---

We can use multiple estimators, each with a different hash function, and then average the produced estimations. Since a single estimator can deviate significantly from the true value (on either side), averaging multiple estimators can reduce this deviation (hence the name "probabilistic counting with *stochastic averaging*").

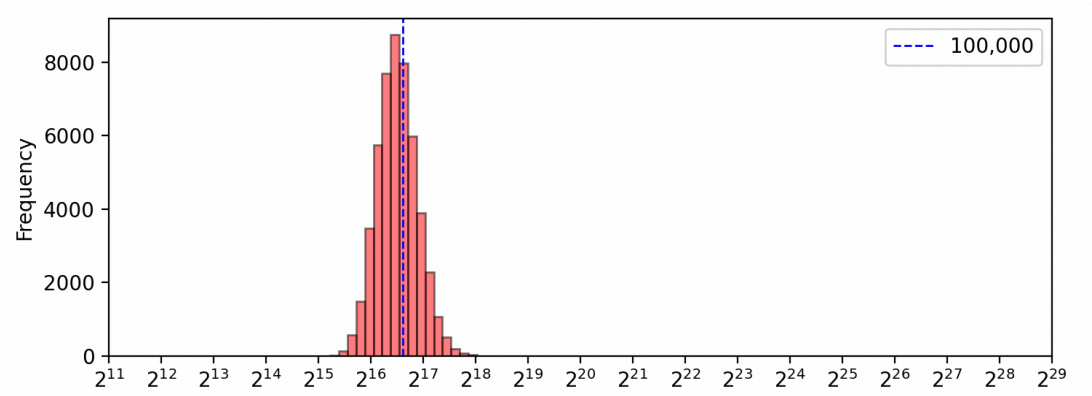
Another idea, which is less expensive than computing multiple hash values is to subdivide the stream into buckets, find the maximum number of trailing 1s in each bucket, and then take the average over all the buckets (some up-scaling is needed). An easy way to subdivide the stream is to use the first k bits of the hash values. For example, elements whose hash values start with `00` are grouped together, elements whose hash values start with `01` are grouped together, etc. If we use the first 2 bits, we get 4 buckets.



The following is a sketch of the modified algorithm.

```
m[M] = {0}                          # array of M=2^k counters initialized to 0
FOREACH x in the stream:
     h = HASH(x)                    # 32-bit integer hash value of x
     c = COUNT-TRAILING-ONES(h)     # e.g. 1010111 has 3 trailing 1s
     d = GET-FIRST-BITS(h, k)       # get the first k bits of h as an integer
     m[d] = MAX(m[d], c)            # update m[d] if c > m[d]

RETURN M * 2^average(m) / 0.77351
```

This algorithm is called PCSA (probabilistic counting with stochastic averaging). The histogram below shows the effect of such a technique on the variance of the estimations (using 16 counters). This algorithm clearly reduces the variance of the produced estimates (compare this histogram to the one before).



Flajolet and his co-authors introduced several improvements to this basic algorithm over the years (LogLog, Super-LogLog, Hyper-LogLog – more on these algorithm names later!). For example, one improvement is to use the harmonic mean instead of the arithmetic mean. Another improvement is to discard the maximum 30 % of the counters. These two improvements reduce the effect of outliers (but they require correction factors). Each improvement (along with its correction factor) was analyzed theoretically and validated experimentally.

Using such improvements, one can get an error rate of around $1.04/\sqrt{M}$, where M is the number of counters used. For example, we can get an error rate of around 3 % using 1024 counters ($1.04/\sqrt{1024} = 0.0325$).

Analysis of Memory Requirements.

The proposed solution uses M counters. Each counter stores a number of trailing 1's in the hash values of the elements. In an application with $n = 1$ billion ($\approx 2^{30}$) unique elements, the maximum number of possible trailing 1's is $\log_2 2^{30} = 30$. To store the number 30, we need $\log_2 30 \approx 5$ bits. In other words, we need $\log_2 \log_2 n$ bits for each counter. Given that the number of counters is constant relative to n, the algorithm requires $O(\log \log n)$ bits overall. This is where the names LogLog and HyperLogLog come from.

The HyperLogLog algorithm is now a standard algorithm implemented in many libraries and used in big tech companies like Google, Twitter and Facebook. For example, Reddit uses HyperLogLog to estimate the number of views a post receives. According to Reddit, the algorithm meets their requirements, where a user must be counted only once within a short time window, counting must be done in real time, and the "the displayed count must be within a few percentage points of the actual tally". Using HyperLogLog, Reddit was able to achieve its requirements using only 0.15 % of the space required to store identifiers for all unique viewers.

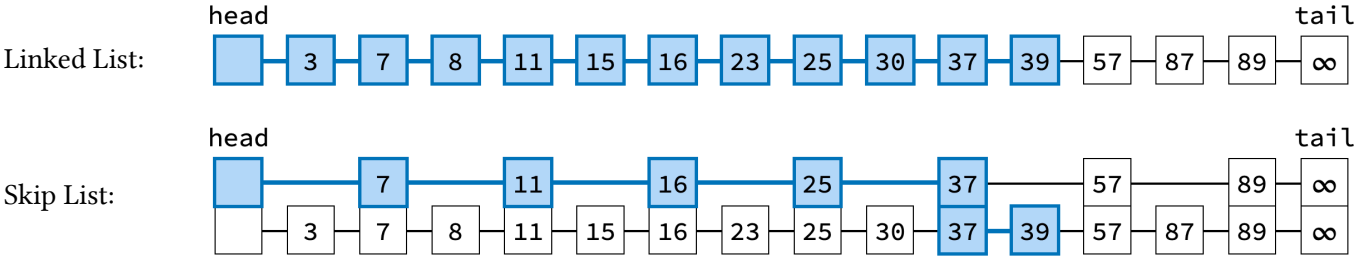
Skip Lists.

A set of n elements implemented as a balanced binary search tree supports search, insertion, and deletion in $O(\log n)$ time. However, insertion and deletion require "rotations" for self-rebalancing that can affect large portions of the tree. Therefore, "locking" of large portions of the tree is needed in multi-threaded applications, which can have a significant effect on the performance of such applications. On the other hand, linked lists are simple and require locking only a couple of nodes during insertion and deletion. However, the searching takes $O(n)$ time, which is impractical.

A Skip List is a randomized data structure that offers the best of the two worlds. It is very simple (compared to a balanced binary search tree), requires locking of only small parts of the data structure during insertions and deletions, and supports search, insertion and deletion in $O(\log n)$ expected time, with high probability.

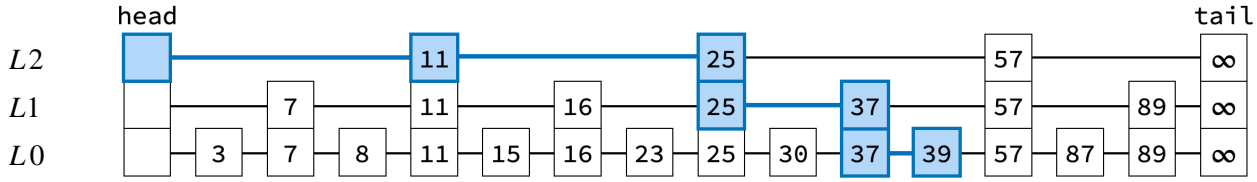
An Ideal Skip List.

Consider the following linked lists containing the same elements (the blue color shows the search path to 39).

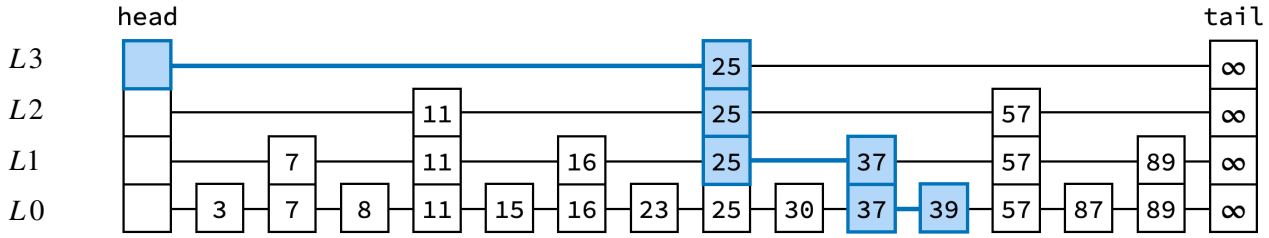


Both lists are sorted. Some of the elements are duplicated in the skip list, allowing to *skip* over some of the elements during the search process. Hence, finding 39 in the skip list requires visiting 6 nodes only, compared to 11 nodes in the standard linked list. The search algorithm in the skip list is simple: Search in the top level until the next node is larger than the element you are searching for (when the next node is 57 in the above example). When that happens, descend to the lower level and move forward until the element is found, or until the next node is larger (i.e. the element is not found).

Note that search in this skip list still runs in $O(n)$, as we might still visit $n/2$ of the elements. However, we can push the idea further and add another level allowing us to skip over some of the elements in the top level. This makes finding 39 take 4 steps instead of 6.



Nothing stops us from adding even another level, which makes finding 39 take 3 steps only.



Searching in in this skip list is similar to doing *binary search*, where moving to the right in $L3$ skips over the first half of the list (i.e. elements < 25), and moving from $L3$ down to $L2$ skips over the second half of the list

(i.e. elements ≥ 25). Similarly, moving to the right in $L2$ allows skipping over $n/4$ elements and moving in $L1$ allows skipping over $n/8$ elements.

We can generalize this idea to build the *ideal skip list* for any n elements. The procedure would be to have all the elements sorted at $L0$, and then to duplicate $n/2$ of the elements in $L1$, $n/4$ of the elements in $L2$, and so on, until the last level gets only one element. Such a list would have (assuming n is a power of 2 for simplicity):

Level	0	1	2	3	…				
# of elements	n	$n/2$	$n/4$	$n/8$	…	8	4	2	1
Power of 2	$2^{\log_2 n}$	$2^{(\log_2 n)-1}$	$2^{(\log_2 n)-2}$	$2^{(\log_2 n)-3}$	…	2^3	2^2	2^1	2^0

In other words, the number of levels in this structure is $\log_2 n + 1$ and the number of nodes is:

$$\sum_{i=0}^{\log_2 n} 2^i \;=\; 2n - 1 = \Theta(n)$$

This tells us that we can achieve $O(\log n)$ search time while still using memory that is linear in the number of elements. However, the difficult question is: How do we insert elements from such a structure?

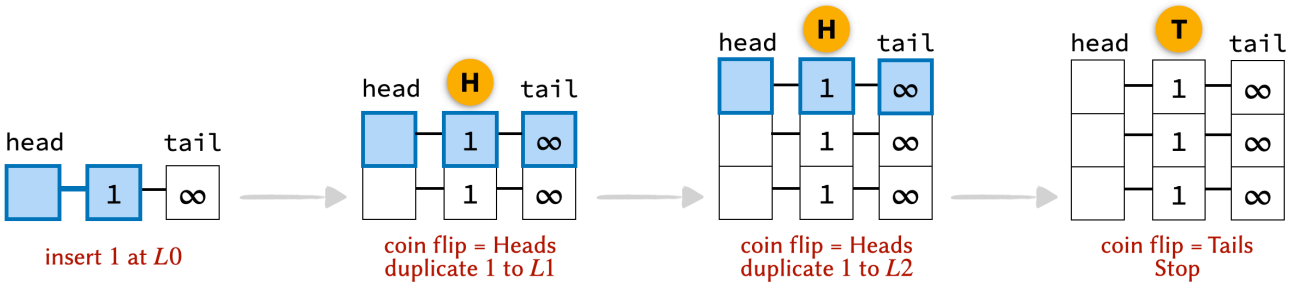
Before answering this question, take a look at the following sketch implementation of class `SkipNode` and the `find` operation, which shows how simple the data structure and the search operation are[1].

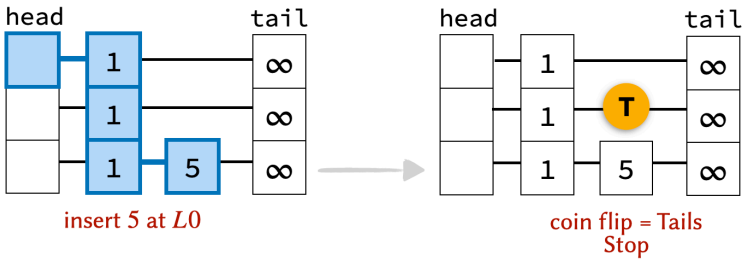
```
class SkipNode {
    Key key             // key (for searching)
    Value value         // value (used by the application)
    SkipNode[] next     // array of next pointers (depending on the level)

    SkipNode(Key x, Value v, int size) { // constructor (given node size)
        key = x; value = v; next = new SkipNode[size];
    }
}


Value find(Key x) {
    int i = topmostLevel            // start at the topmost nonempty level
    SkipNode p = head               // start at the head node

    while (i >= 0) {                // while levels remain
        if (p.next[i].key <= x)
            p = p.next[i]           // advance along the same level
        else
            i--                     // drop down a level
    }

    if (p.key == x) return p.value  // return value if found
    else            return null
}
```

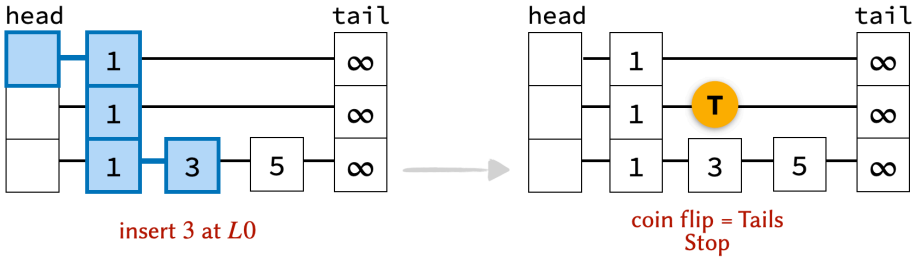
[1] These are (syntax highlighted) screenshots from Dave Mount's CMS420 lecture notes on Skip Lists

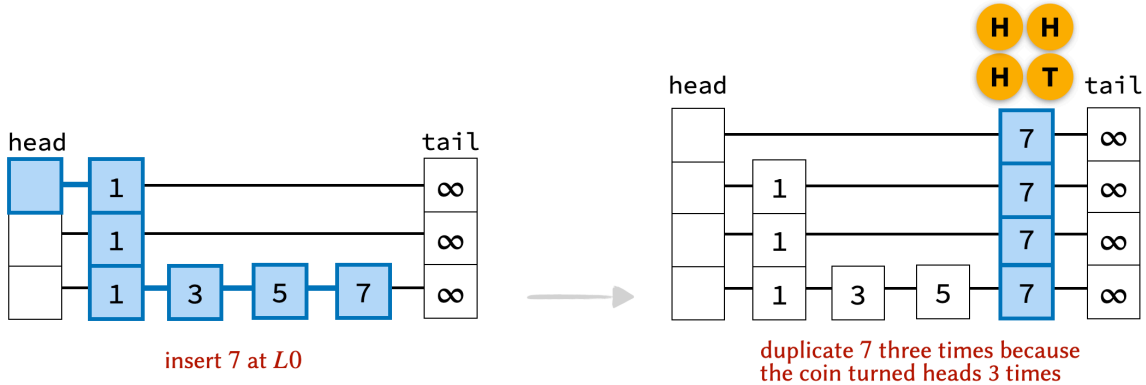
A Randomized Skip List.

To make insertion easy for us, we will follow the element search path until we get to $L0$, which is where we will insert the element. Next, we will flip a coin. If the coin turns heads, we will duplicate the element to the upper level (and update the links as appropriate). We will repeat this until the coin turns tails. Each time the coin turns heads, the node is duplicated to the upper level (creating a new level if necessary). The following example shows the procedure for inserting the elements 1, 5, 3, 2, 7 starting from an empty list.









This insertion procedure is simple, but does it produce a list that consumes $\Theta(n)$ memory and that has a worst case search time of $\Theta(\log n)$? Theoretically speaking, we can keep getting heads each time we flip a coin (forever). What we are interested in is the typical behavior and the probability of seeing this typical behavior.

Theorem. Given a randomized skip list containing n elements, the probability that the number of levels is greater than $c \log n$ is: $\frac{1}{n^c - 1}$, where c is a constant greater than 1.

Proof. For any inserted element x, it is duplicated to $L1$ with probability 1/2 and to $L2$ with probability $(1/2)^2$ and to L_i with probability $(1/2)^i$. Therefore, the probability of x being duplicated to a level higher than $c \log_2 n$ (where $c \geq 1$) is less than:

$$\frac{1}{2^{c \log_2 n}} = \frac{1}{(2^{\log_2 n})^c} = \frac{1}{n^c}$$

The probability that either the 1st element, the 2nd, or the 3rd, etc. is duplicated to a level higher than $c \log_2 n$ is less than the sum of their individual probabilities, which makes the total = $n \times \frac{1}{n^c} = \frac{1}{n^{c-1}}$

To appreciate how low this probability is, let's consider $c = 3$. The probability that the number of levels exceeds $3 \log_2 n$ is less than 10^{-6} if $n = 1000$ and less than 10^{-8} if $n = 10000$. In other words, the probability that the number of levels in the randomized skip list exceeds $\Theta(\log n)$ is too low to be of concern.

Despite the above analysis, one can still argue that while the number of levels is $O(\log n)$, we might end up with $\Theta(n)$ nodes in each of the $O(\log n)$ levels, making the memory used $\Theta(n \log n)$ or the skipping not useful enough to reduce the search time to $O(\log n)$. This is a valid concern, which we address in the following analysis.

Expected Memory Requirements. The expected amount of memory used by a skip list is the sum of the expected memory used in every level. Let L be the maximum level in the skip list and $E(i)$ be the expected amount of memory used at level i, we are interested in the value of the following sum:

$$\sum_{i=0}^{L} E(i)$$

Since an element has probability $(1/2)^i$ of being duplicated up to level i, the expected number of elements that are at level i is $E(i) = n/2^i$. This makes the above sum as follows:

$$\sum_{i=0}^{L} \frac{n}{2^i} = n \sum_{i=0}^{L} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2n$$

In other words, the expected amount of used memory is still linear.

Expected Search Time. Let's assume that element x was found in the skip list. Observe that if we trace our way *backward* from x along the search path, we move up if there is a copy above the node, and move left otherwise. Convince yourself that this is always true.

Since each node at level i had a 50% chance of being duplicated to the level above during the insertion process, there is a 50% chance that x has a copy above it (i.e. the search path comes from above) and a 50% chance that it does not (i.e. the search comes from left). Regardless of where the search comes from, the same reasoning can be applied again, giving us the following recurrence for the expected number of steps done starting at level i :

$$T(i) = 1 + \tfrac{1}{2}T(i-1) + \tfrac{1}{2}T(i)$$

We can simplify this recurrence as follows:

$$\begin{aligned} 2T(i) &= 2 + T(i-1) + T(i) \\ T(i) &= 2 + T(i-1) \end{aligned}$$

Expanding the recurrence, we get $T(i) = 2 + 2 + 2 + \ldots$ repeated $L + 1$ times, where L is the maximum level. Since the expected number of levels is $\leq c \log_2 n$ with high probability, the number of steps $T(i)$ is expected to be $\leq 2c \log_2 n$.

Randomization in Algorithms & Data Structures
Part 2

Ibrahim Albluwi

Matrix-Product Verification.

Given three matrices A, B and C, of size $n \times n$ each, verify that $AB = C$.

A straightforward way to approach this problem is to multiply A and B and check if the result is C. A naive implementation of this solution runs in $\Theta(n^3)$. Using more clever matrix multiplication algorithms can give better running times, but the best-known ones run in $\omega(n^{2.37})$.

Freivald's algorithm is a randomized algorithm whose basic implementation runs in $\Theta(n^2)$ and produces a correct answer with a very high probability. The algorithm can be made even more efficient using ideas that we won't cover here.

https://www.jgindi.me/posts/check-matmul/

https://cse.iitkgp.ac.in/~swagato/matprod.pdf

https://en.wikipedia.org/wiki/Freivalds%27_algorithm

Krager's Algorithm for finding the Global Min-Cut.

Whatever

Johnson's Algorithm for MAX 3SAT.

Whatever